Intelligent Search Motion Planning in a Warehouse

Group 50

Adrian Ash, Chiran Walisundara and Don Kaluarachchi

Introduction

Sokoban is a computer/mobile game puzzle where the player moves boxes/crates about a warehouse in an attempt to move them to storage/goal areas. The game is played on a grid of squares, with each square representing a wall or floor. Some floor squares include boxes, while others are designated as storage areas. The player is restricted inside the board and can only move vertically or horizontally onto vacant squares. The player however cannot move through a box. A box may be moved by the player by stepping up to it and pushing it to the next square. Boxes cannot be pulled or pushed into squares with other boxes or walls. There is an equal number of boxes and storage locations on the grid while a round is completed when all boxes are kept in storage locations. This motivated problem can be fed into an artificial intelligence algorithm to have optimistic solutions to rigorous human searches.

State Representations & Heuristics

The Sokoban puzzle contains certain key elements that have a role in solving the puzzle. These are,

- Worker location
- Box locations
- Wall locations
- Goal locations

These can be categorised as static/dynamic and can be part of the problem instance or part of the state.

Static elements do not need to be modified at any point in order to finish the problem, but dynamic elements do need to be updated for the puzzle to be completed.

A state representation is a way that the real-world configuration is modelled in the Sokoban problem. The states of the Sokoban problem are the worker location and box locations. These are extracted from the warehouse object in the Sokoban solver and are represented in the form of tuples of tuples where the first tuple is the location of the worker, and the following represents the location of the boxes. These need to be updated for the puzzle to be completed therefore they are classified as dynamic.

An example of a state would be:

Worker location = (4, 6)
Box Locations = (2, 3) and (3, 4)
Then the state = ((4, 6), (2, 3), (3, 4))

On the contrary, those classified as static are elements that do not change and need not be updated. These are elements such as the locations of the walls and goals which are part of the problem instance.

To answer the question of whether a warehouse is a good state representation, it can be concluded that it is a good representation as a warehouse allows for better grid/graph representations of the Sokoban problem. This is beneficial in this instance over a tree search as we iterate over multiple paths to find the most optimal path from player to boxes and boxes to goals.

A heuristic is a technique used in artificial intelligence taking into consideration all possible outcomes and finding an optimal solution. In the context of AI, a heuristic function at each branching step evaluates the available information/actions and evaluations the decision to make and which branch to follow.

In this search problem, the heuristic function given by h(n) takes in a node object and returns the best action the worker can take to reach the goal state. The heuristic is defined as,

- The manhattan distance from the worker to the box
- The minimum manhattan distance from the box to a goal (shortest path to goal)
- The maximum combined values of the two values mentioned above are returned
- Boxes that have reached the goal state are not checked

This heuristic, h(n), is admissible as it will always be less than or equal to the true cost to the goal state, h*(n). This is guaranteed using the minimum distance from the box to the goal. The goal state in the worst case will always be at least the minimum distance from the box to the goal.

It is important to note that our heuristic is not consistent. This is our problem has multiple-goal state objectives (getting each box onto a target). After exploring multiple different heuristics, however, it was found that treating our problem as multiple different sub-problems (focus on a single box to the goal at a time) was the best solution as the trade-off of a more consistent heuristic significantly increased the run time.

Both will be explained in further detail in the "Performance, Limitations & Improvements of the solver" section of this report.

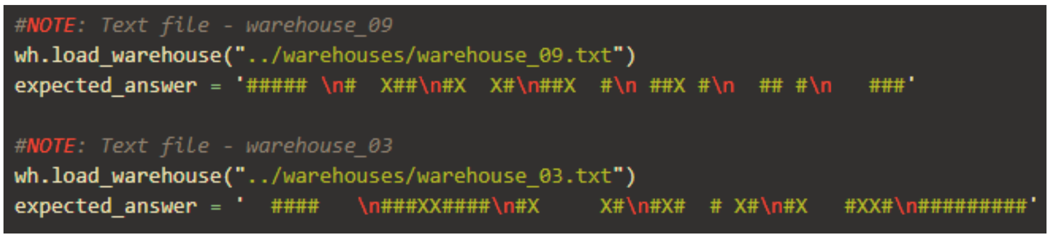
Testing Methodology

Unit tests should be implemented in code to ensure that an isolated part of the code performs as intended. In this implementation of the Sokoban, the sanity check file provided was used as the main tool for testing.

The file contains in it multiple functions that run the functions in the "mySokobanSolver.py". The functions present in the "sanity_check.py" are as follows,

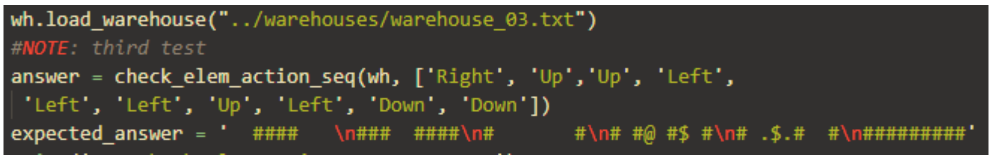
- test_taboo_cells
- test_check_elem_action_seq
- test_solve_weighted_sokoban

Multiple extra tests were added to these original functions. Figures 1, 2 and 3 show added tests for each of the above functions respectively:



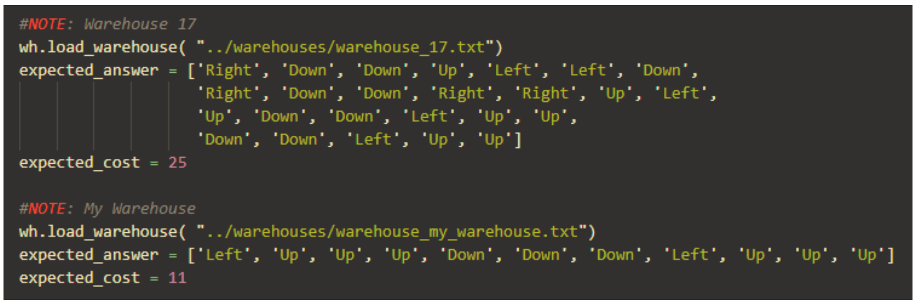
```python
#NOTE: Text file - warehouse_09
wh.load_warehouse("../warehouses/warehouse_09.txt")
expected_answer = '##### \n#  X##\n#X  X#\n##X  #\n ##X #\n  ## #\n   ###'

#NOTE: Text file - warehouse_03
wh.load_warehouse("../warehouses/warehouse_03.txt")
expected_answer = '  ####   \n###XX####\n#X     X#\n#X#  # X#\n#X   #XX#\n#########'
```

Figure 1 – Extra warehouses used in the test_taboo_cells() function for testing



```
wh.load_warehouse("../warehouses/warehouse_03.txt")
#NOTE: third test
answer = check_elem_action_seq(wh, ['Right', 'Up','Up', 'Left',
 'Left', 'Left', 'Up', 'Left', 'Down', 'Down'])
expected_answer = '  ####    \n###   ####\n#        #\n# #@ #$ #\n# .$.#  #\n#########'
```

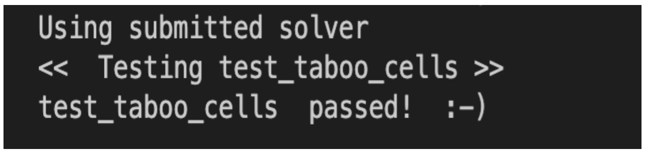
Figure 3 - Extra warehouse used in the test_check_elem_action_seq() function for testing



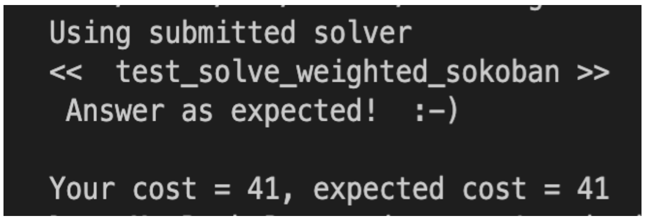
```
#NOTE: Warehouse 17
wh.load_warehouse( "../warehouses/warehouse_17.txt")
expected_answer = ['Right', 'Down', 'Down', 'Up', 'Left', 'Left', 'Down',
                   'Right', 'Down', 'Down', 'Right', 'Right', 'Up', 'Left',
                   'Up', 'Down', 'Down', 'Left', 'Up', 'Up',
                   'Down', 'Down', 'Left', 'Up', 'Up']
expected_cost = 25

#NOTE: My Warehouse
wh.load_warehouse( "../warehouses/warehouse_my_warehouse.txt")
expected_answer = ['Left', 'Up', 'Up', 'Up', 'Down', 'Down', 'Down', 'Left', 'Up', 'Up', 'Up']
expected_cost = 11
```

Figure 2 - Extra warehouses used in the test_solve_weighted_sokoban() function for testing

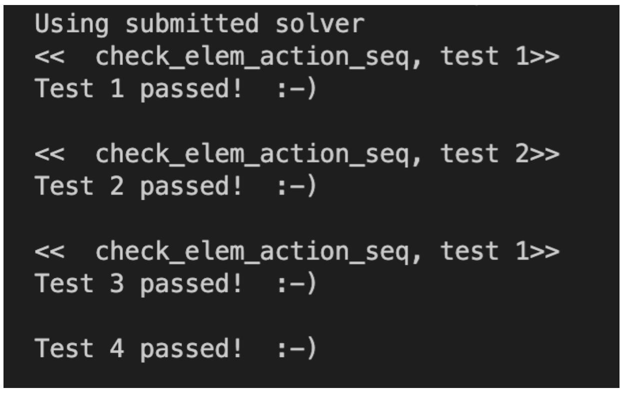
The testing method implemented in each of these functions uses a string comparison. That is for each of the warehouses a present string is used - which is calculated by hand, this is then compared with the output of said function which returns a string. If the strings match the function has passed the tests. Examples of this running for a particular warehouse are shown in the figures below (Figures 4, 5 and 6).



```
Using submitted solver
<<  Testing test_taboo_cells >>
test_taboo_cells  passed!  :-)
```



```
Using submitted solver
<<  test_solve_weighted_sokoban >>
 Answer as expected!  :-)

Your cost = 41, expected cost = 41
```



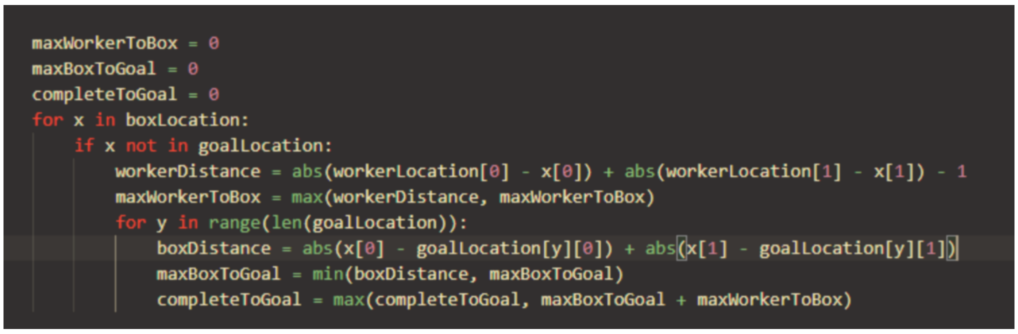
```
Using submitted solver
<<  check_elem_action_seq, test 1>>
Test 1 passed!  :-)

<<  check_elem_action_seq, test 2>>
Test 2 passed!  :-)

<<  check_elem_action_seq, test 1>>
Test 3 passed!  :-)

Test 4 passed!  :-)
```

Figure 4, 5 and 6 - Each figure shows the results of multiple different testing functions

Performance, Limitations & Improvements of Solver

The parameters to determine the efficiency of the heuristic are given by two parameters, these are mainly admissibility and consistency.

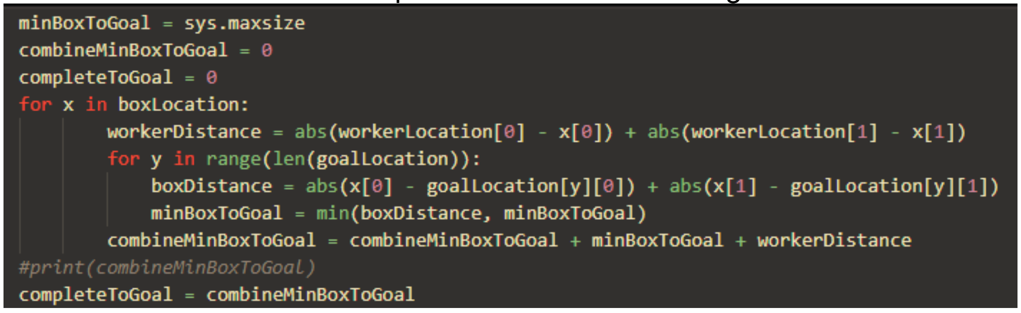
The admissibility condition is met when the heuristic never overestimates the true cost to the nearest goal. The consistent condition is met when going from neighbouring nodes a to b, the heuristic difference/path cost never overestimates the actual path cost.



```
maxWorkerToBox = 0
maxBoxToGoal = 0
completeToGoal = 0
for x in boxLocation:
    if x not in goalLocation:
        workerDistance = abs(workerLocation[0] - x[0]) + abs(workerLocation[1] - x[1]) - 1
        maxWorkerToBox = max(workerDistance, maxWorkerToBox)
        for y in range(len(goalLocation)):
            boxDistance = abs(x[0] - goalLocation[y][0]) + abs(x[1] - goalLocation[y][1])
            maxBoxToGoal = min(boxDistance, maxBoxToGoal)
            completeToGoal = max(completeToGoal, maxBoxToGoal + maxWorkerToBox)
```

As shown by the image above multiple other heuristics were implemented to compare the benefits and drawbacks of each. An earlier version of the heuristic function had a condition where the box to goal state was set to zero which produced a fast solver but was unable to solve complex problems.

The optimal heuristic function for this problem is shown in the figure below.



```
minBoxToGoal = sys.maxsize
combineMinBoxToGoal = 0
completeToGoal = 0
for x in boxLocation:
        workerDistance = abs(workerLocation[0] - x[0]) + abs(workerLocation[1] - x[1])
        for y in range(len(goalLocation)):
            boxDistance = abs(x[0] - goalLocation[y][0]) + abs(x[1] - goalLocation[y][1])
            minBoxToGoal = min(boxDistance, minBoxToGoal)
        combineMinBoxToGoal = combineMinBoxToGoal + minBoxToGoal + workerDistance
#print(combineMinBoxToGoal)
completeToGoal = combineMinBoxToGoal
```

During the process of building the heuristic, the speed and complexity were weighed against each other and the final heuristic implemented was optimal for those parameters. The heuristic meets the admissibility condition but does not always meet the consistent condition. This is due to the problem of multiple goals. That is when the worker moves toward one box to get it to the goal state it sometimes has to move away from the other box and its goal.

There is a handful of warehouses that return an impossible result since a solution does not exist for that particular case where the player can't successfully transport all the boxes into the goal locations. There isn't a solution from the heuristic point of view that can be implemented for this issue as it isn't a limitation.

Some warehouses take too long to calculate the solution which makes for slight time delays in delivery. This is because the current heuristic doesn't keep track of boxes that have reached the goal. However, there are instances where a box that has reached a goal has to be moved to set down another box to its goal as it obstructs the path.

To mitigate/avoid this limitation, the heuristic can be improved to calculate the distance to all the boxes at each time iteration. This way the solution will be much more admissible but ultimately will take even more time than the original solution since more memory is being stored at each iteration.

Another solution to this problem is by relaxing the admissibility condition which can speed up the solution search at the price of optimality through the estimation of shorter routes. The current method can be modified to A* epsilon by using the first heuristic as the 'focal' list, which is used to select candidate nodes, and the second heuristic is used to select the most promising node from the 'focal' list.